What is Remix?

Remix is a powerful, web-based Integrated Development Environment (IDE) specifically designed for writing, deploying, testing, and debugging Ethereum smart contracts. It is widely used by developers for its ease of use and feature-rich interface, making it an ideal tool for both beginners and experienced professionals.

Key features of Remix:

- **Web-Based:** No installation is required; it runs directly in your browser.
- **Built-In Solidity Compiler:** Easily compile your smart contracts without additional setup.
- **Deploy and Test:** Deploy contracts to test networks or the Ethereum mainnet and interact with them in real time.
- **Debugging Tools:** Step-by-step debugging for tracing and resolving issues in your smart contract code.
- **Plugin Ecosystem:** Additional plugins for linting, security analysis, and more.

What Are Other Tools for Testing Smart Contracts?

Besides Remix, there are several other tools available for testing and deploying Ethereum smart contracts:

1. **Hardhat:**
    - A JavaScript-based development environment.
    - Offers local Ethereum nodes, advanced debugging, and script automation.
    - Ideal for large projects with more complex workflows.
2. **Truffle:**
    - A popular framework for developing and testing contracts.
    - Provides migrations, testing, and a built-in Ethereum client (Ganache).
    - Well-suited for projects that need structured deployment pipelines.
3. **Ganache:**
    - A local Ethereum blockchain simulator.
    - Allows you to test your contracts in a local environment without incurring gas fees.
    - Often used in combination with Truffle or Hardhat.
4. **Foundry:**
    - A modern toolchain for Ethereum development.
    - Fast and lightweight, it includes features for testing, fuzzing, and scripting.
    - Written in Rust, it's popular among developers focusing on performance.

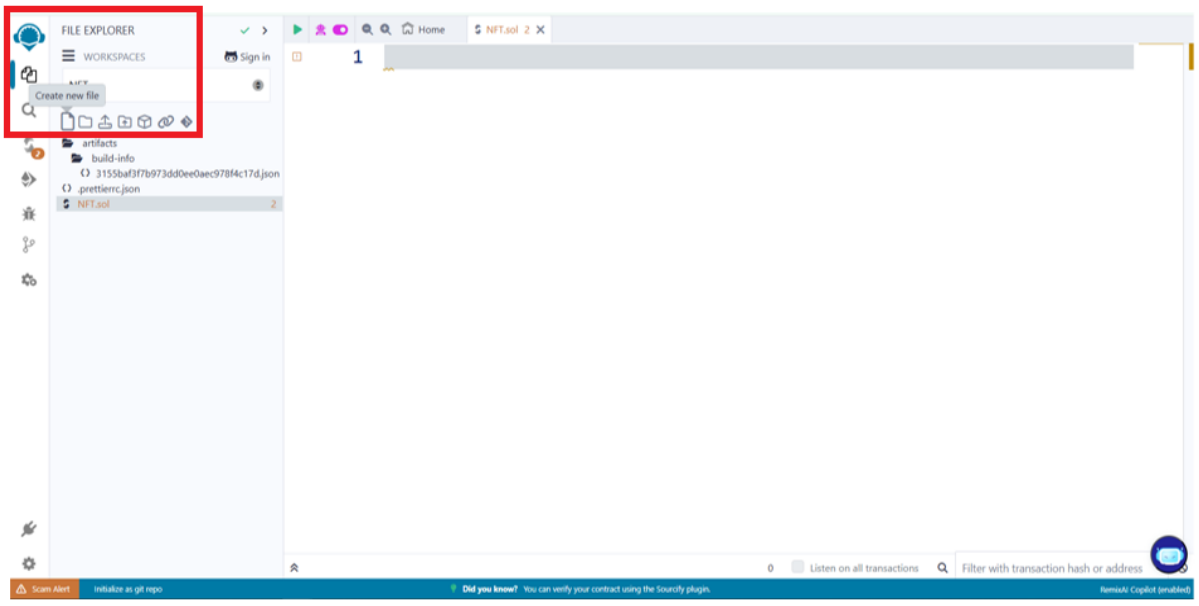
Why Did we Choose Remix?

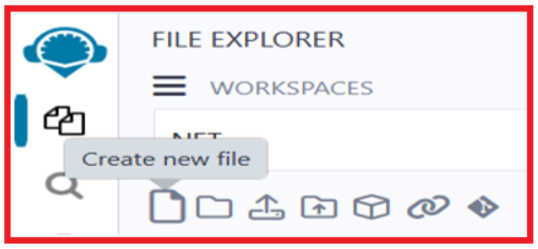
We chose Remix for this project because of its simplicity and accessibility. It allows me to focus on writing and testing smart contracts without spending extra time on setup or configuration. Some specific reasons include:

1. **Ease of Use:** The web-based interface eliminates the need for installations and setup, making it easy to get started immediately.

2. **All-in-One Environment:** Remix integrates compiling, deploying, testing, and debugging within a single platform.
3. **Immediate Feedback:** The interface provides real-time feedback on errors and execution results, which is crucial for refining and debugging smart contracts.
4. **Beginner-Friendly:** Remix is ideal for quick prototyping and testing small contracts like the one I am working on here.

Creating a New File in Remix IDE and Starting a Smart Contract


FILE EXPLORER
WORKSPACES
Sign in
Create new file
artifacts
build-info
3155baf3f7b973dd0ee0aec978f4c17d.json
.prettierrc.json
NFT.sol
Home
NFT.sol
Listen on all transactions
Filter with transaction hash or address
Scam Alert
Initialize as git repo
Did you know? You can verify your contract using the Sourcify plugin.
RemixAI Copilot (enabled)



The screenshot showcases the **File Explorer** panel in Remix IDE, where we start setting up our project by creating a new file for the smart contract.

1. File Explorer Panel

- Located on the left side of the interface, this panel helps organize and manage all files in your workspace.

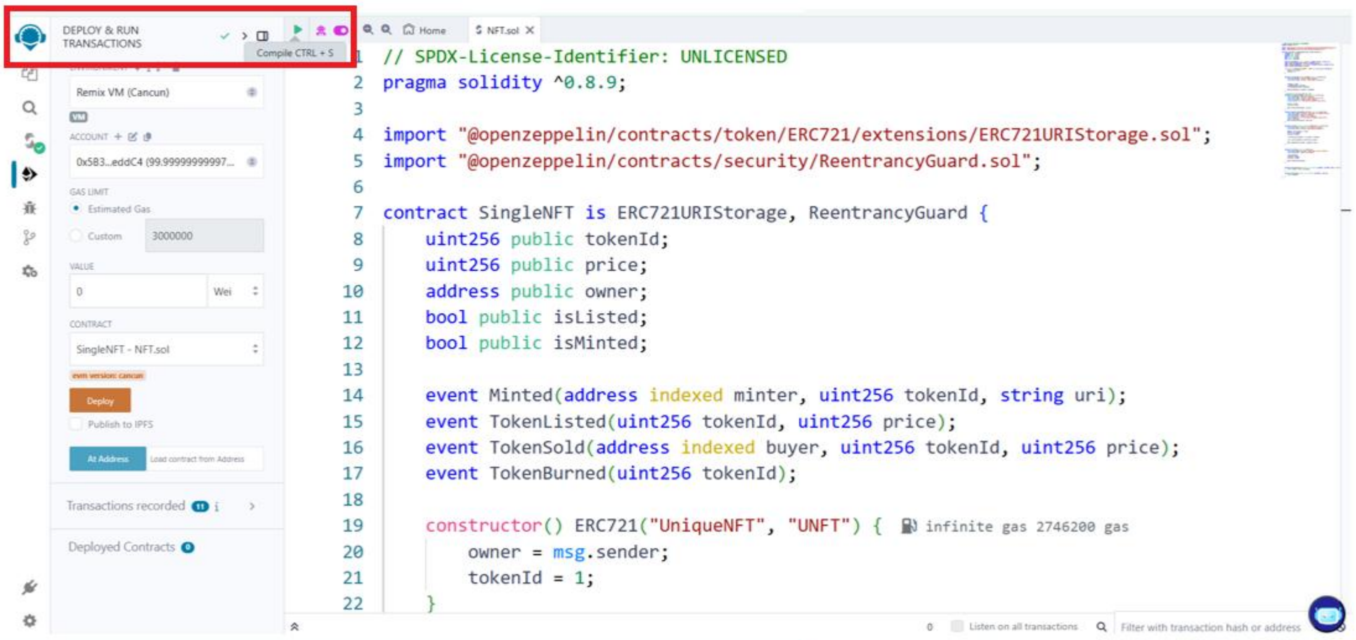
2. Create New File Button

- Highlighted in the red box, the "Create New File" button (with a document icon) allows you to create a new file in the project.
    - Click the button to open a dialog box where you can name the file.
    - Example: Name the file NFT.sol for Solidity-based contracts.

3. Starting the Smart Contract

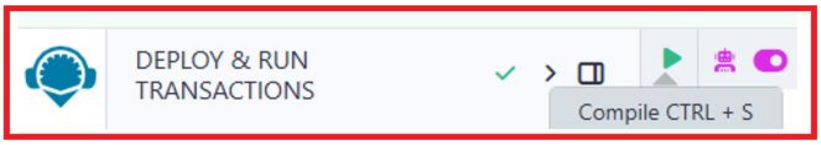
- Once the file is created, it will appear in the File Explorer under your current workspace.
- Click on the new file to open it in the editor, where you can begin writing the smart contract code.
- In this case, we'll start with a basic contract structure, which will be expanded upon step by step.

Adding Code and Compiling in Remix IDE


DEPLOY & RUN TRANSACTIONS
Compile CTRL + S
Home
NFT.sol
Deploy
Publish to IPFS
At Address
Load contract from Address
Transactions recorded 11
Deployed Contracts 0
Listen on all transactions
Filter with transaction hash or address

Remix VM (Cancun)
ACCOUNT 0x5B3...eddC4 (99.99999999997...
GAS LIMIT ● Estimated Gas ○ Custom 3000000
VALUE 0 Wei
CONTRACT SingleNFT - NFT.sol
evm version: cancun

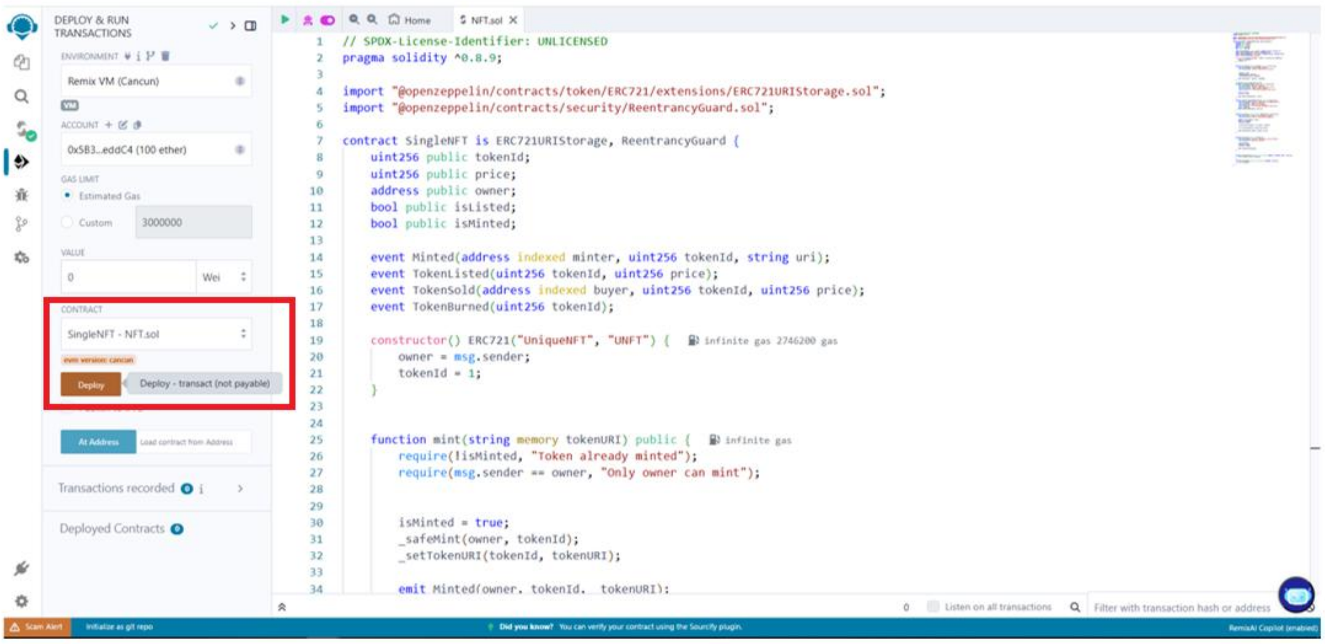
```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SingleNFT is ERC721URIStorage, ReentrancyGuard {
    uint256 public tokenId;
    uint256 public price;
    address public owner;
    bool public isListed;
    bool public isMinted;

    event Minted(address indexed minter, uint256 tokenId, string uri);
    event TokenListed(uint256 tokenId, uint256 price);
    event TokenSold(address indexed buyer, uint256 tokenId, uint256 price);
    event TokenBurned(uint256 tokenId);

    constructor() ERC721("UniqueNFT", "UNFT") {  infinite gas 2746200 gas
        owner = msg.sender;
        tokenId = 1;
    }
```



The screenshot shows that we have moved to the **Deploy & Run Transactions** panel in Remix IDE and demonstrates the process of compiling the smart contract.

1. **Adding the Code**:
    - The Solidity code for the SingleNFT contract has been written in the editor.
    - This contract includes functionalities for minting, listing, selling, and burning an NFT. Each function will be explained step by step in the subsequent sections.
2. **Compiling the Contract**:
    - After entering the code, the contract needs to be compiled to check for errors and generate the necessary bytecode and ABI for deployment.
    - Click the **Compile** button (highlighted in the lower panel) or use the shortcut CTRL + S to trigger compilation.
    - If the code is error-free, the green checkmark appears next to the contract name in the Compile tab, indicating successful compilation.
3. **Deploy & Run Transactions Panel**:
    - In this panel, we select:
        - **Environment**: The simulation environment (Remix VM in this case).
        - **Contract**: The compiled contract (SingleNFT).
        - **Gas Limit** and **Value**: Optional parameters for deployment and transactions.
4. **Next Steps**:
    - The successfully compiled contract is now ready for deployment and interaction.
    - The following sections will explain the functions and features of the SingleNFT contract in detail.

Deploying the Smart Contract in Remix IDE



DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Remix VM (Cancun)

ACCOUNT: 0x5B3...eddC4 (100 ether)

GAS LIMIT: Estimated Gas; Custom 3000000

VALUE: 0 Wei

CONTRACT: SingleNFT - NFT.sol

evm version: cancun

Deploy | Deploy - transact (not payable)

At Address | Load contract from Address

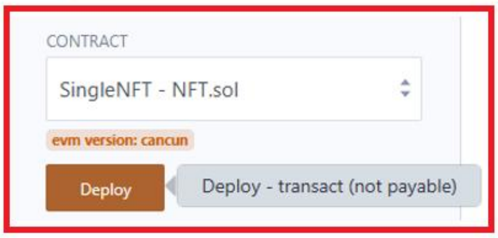
Transactions recorded

Deployed Contracts

NFT.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SingleNFT is ERC721URIStorage, ReentrancyGuard {
    uint256 public tokenId;
    uint256 public price;
    address public owner;
    bool public isListed;
    bool public isMinted;

    event Minted(address indexed minter, uint256 tokenId, string uri);
    event TokenListed(uint256 tokenId, uint256 price);
    event TokenSold(address indexed buyer, uint256 tokenId, uint256 price);
    event TokenBurned(uint256 tokenId);

    constructor() ERC721("UniqueNFT", "UNFT") {    infinite gas 2746200 gas
        owner = msg.sender;
        tokenId = 1;
    }


    function mint(string memory tokenURI) public {    infinite gas
        require(!isMinted, "Token already minted");
        require(msg.sender == owner, "Only owner can mint");


        isMinted = true;
        _safeMint(owner, tokenId);
        _setTokenURI(tokenId, tokenURI);

        emit Minted(owner, tokenId, tokenURI);
```

Did you know? You can verify your contract using the Sourcify plugin.



CONTRACT

SingleNFT - NFT.sol

evm version: cancun

Deploy | Deploy - transact (not payable)

The screenshot shows the process of deploying the SingleNFT contract in Remix IDE using the **Deploy & Run Transactions** panel.

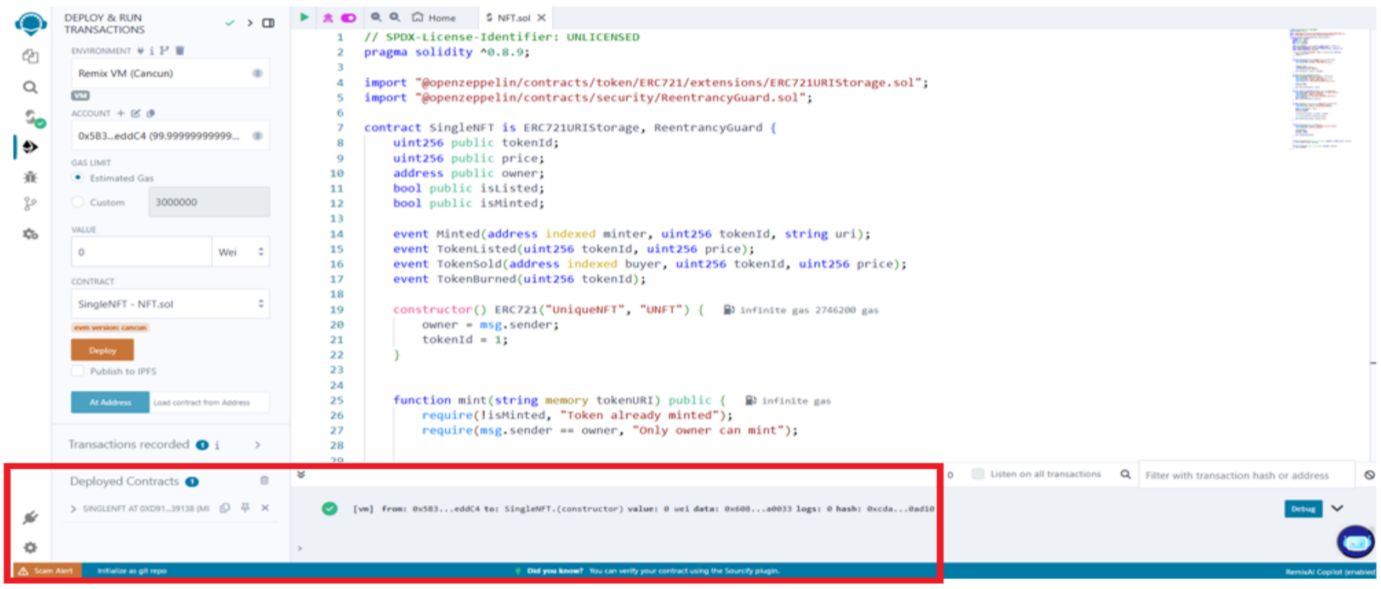
1. **Select the Environment**:
    - In the **Environment** dropdown (top-left of the panel), choose the desired testing environment. In this case, Remix VM (Cancun) is selected, which provides a simulated blockchain environment for testing.
2. **Account and Gas Settings**:
    - **Account**: Select the account to deploy the contract from. The account displayed has 100 ETH for testing.
    - **Gas Limit**: Customize the gas limit if necessary (default is sufficient in most cases).
    - **Value**: Set the value to be sent with the deployment transaction (in this case, it is 0 Wei).
3. **Select the Contract**:
    - In the **Contract** dropdown, choose the contract to deploy. Here, SingleNFT - NFT.sol is selected, which refers to the compiled SingleNFT contract.
    - Ensure the contract is compiled successfully before deploying.
4. **Deploy the Contract**:
    - Click the **Deploy** button to send the deployment transaction to the selected environment.

- ○ The transaction details (e.g., hash, gas used) will appear in the bottom panel once the deployment is successful.

5. **Deployed Contracts Section**:

   - ○ After deployment, the contract will appear in the **Deployed Contracts** section of the panel. From here, you can interact with the contract's functions.

Deployment Results in Remix IDE

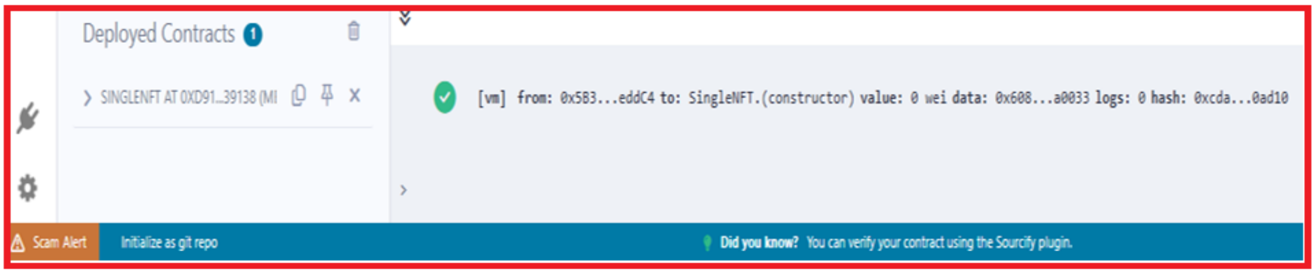

DEPLOY & RUN TRANSACTIONS
Home
NFT.sol
Deploy
Publish to IPFS
At Address
Load contract from Address
Listen on all transactions
Filter with transaction hash or address
Debug
Scam Alert
Initialize as git repo
RemixAI Copilot (enabled)

ENVIRONMENT: Remix VM (Cancun)
ACCOUNT: 0x5B3...eddC4 (99.99999999999...
GAS LIMIT: Estimated Gas ● Custom 3000000
VALUE: 0 Wei
CONTRACT: SingleNFT - NFT.sol
evm version: cancun
Transactions recorded 1
Deployed Contracts 1
SINGLENFT AT 0XD91...39138 (MI

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SingleNFT is ERC721URIStorage, ReentrancyGuard {
    uint256 public tokenId;
    uint256 public price;
    address public owner;
    bool public isListed;
    bool public isMinted;

    event Minted(address indexed minter, uint256 tokenId, string uri);
    event TokenListed(uint256 tokenId, uint256 price);
    event TokenSold(address indexed buyer, uint256 tokenId, uint256 price);
    event TokenBurned(uint256 tokenId);

    constructor() ERC721("UniqueNFT", "UNFT") {    infinite gas 2746200 gas
        owner = msg.sender;
        tokenId = 1;
    }


    function mint(string memory tokenURI) public {    infinite gas
        require(!isMinted, "Token already minted");
        require(msg.sender == owner, "Only owner can mint");
```

```
[vm] from: 0x5B3...eddC4 to: SingleNFT.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0xcda...0ad10
```

Did you know? You can verify your contract using the Sourcify plugin.



Deployed Contracts 1
SINGLENFT AT 0XD91...39138 (MI

```
[vm] from: 0x5B3...eddC4 to: SingleNFT.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0xcda...0ad10
```

Did you know? You can verify your contract using the Sourcify plugin.

The screenshot highlights the results of successfully deploying the SingleNFT contract in Remix IDE, as well as the deployed contract's details in the Deployed Contracts panel.

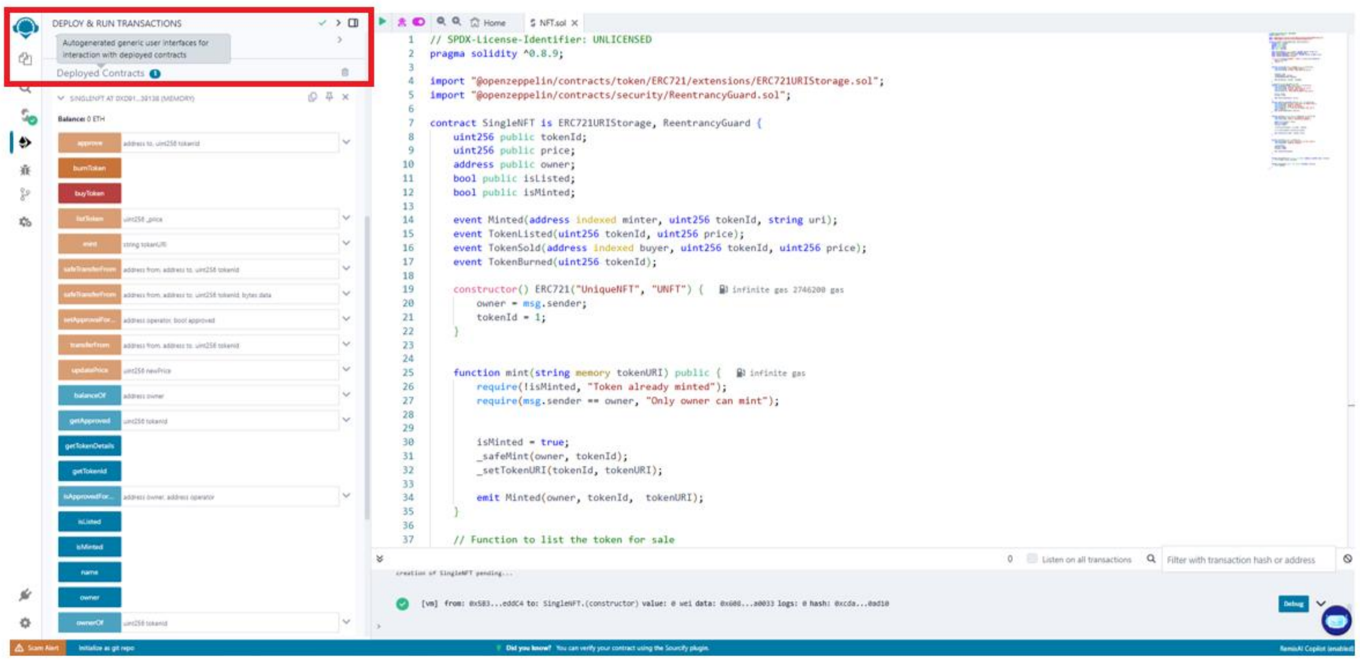
1. **Deployment Confirmation:**

   - ○ After clicking the Deploy button, the transaction for deploying the SingleNFT contract is processed.
   - ○ The bottom panel shows the transaction details, including:
     - ▪ From: The address of the account that deployed the contract.
     - ▪ To: Indicates the deployed contract (SingleNFT).
     - ▪ Value: The amount of Wei sent with the transaction (in this case, 0 Wei).
     - ▪ Gas Used: The gas consumed during deployment.
     - ▪ Transaction Hash: A unique identifier for the deployment transaction.

2. **Deployed Contracts Section:**

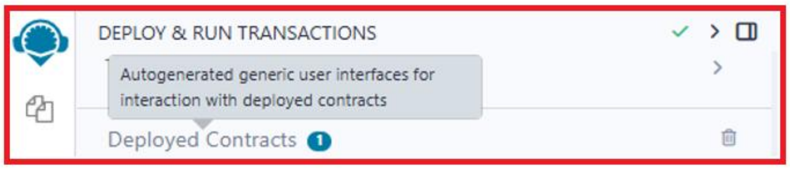
   - ○ The SingleNFT contract appears in the Deployed Contracts panel.
   - ○ From here, you can expand the contract to interact with its functions (e.g., mint, listToken, etc.)

Interacting with the Deployed Contract in Remix IDE


DEPLOY & RUN TRANSACTIONS
Autogenerated generic user interfaces for interaction with deployed contracts
Deployed Contracts 1
SINGLENFT AT 0XD91...39138 (MEMORY)
Balance: 0 ETH
approve
address to, uint256 tokenId
burnToken
buyToken
listToken
uint256 _price
mint
string tokenURI
safeTransferFrom
address from, address to, uint256 tokenId
safeTransferFrom
address from, address to, uint256 tokenId, bytes data
setApprovalFor...
address operator, bool approved
transferFrom
address from, address to, uint256 tokenId
updatePrice
uint256 newPrice
balanceOf
address owner
getApproved
uint256 tokenId
getTokenDetails
getTokenId
isApprovedFor...
address owner, address operator
isListed
isMinted
name
owner
ownerOf
uint256 tokenId
Home
NFT.sol
Listen on all transactions
Filter with transaction hash or address
Debug
Scam Alert
Initialize as git repo
Did you know? You can verify your contract using the Sourcify plugin.
RemixAI Copilot (enabled)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SingleNFT is ERC721URIStorage, ReentrancyGuard {
    uint256 public tokenId;
    uint256 public price;
    address public owner;
    bool public isListed;
    bool public isMinted;

    event Minted(address indexed minter, uint256 tokenId, string uri);
    event TokenListed(uint256 tokenId, uint256 price);
    event TokenSold(address indexed buyer, uint256 tokenId, uint256 price);
    event TokenBurned(uint256 tokenId);

    constructor() ERC721("UniqueNFT", "UNFT") {   infinite gas 2746200 gas
        owner = msg.sender;
        tokenId = 1;
    }


    function mint(string memory tokenURI) public {   infinite gas
        require(!isMinted, "Token already minted");
        require(msg.sender == owner, "Only owner can mint");


        isMinted = true;
        _safeMint(owner, tokenId);
        _setTokenURI(tokenId, tokenURI);

        emit Minted(owner, tokenId,  tokenURI);
    }

    // Function to list the token for sale
```

```
creation of SingleNFT pending...

[vm] from: 0x5B3...eddC4 to: SingleNFT.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0xcda...8ad10
```


DEPLOY & RUN TRANSACTIONS
Autogenerated generic user interfaces for interaction with deployed contracts
Deployed Contracts 1

The screenshot illustrates how to interact with the deployed SingleNFT contract using the **Deploy & Run Transactions** panel in Remix IDE.

1. **Deployed Contracts Section**:
    - After successfully deploying the contract, it appears under the **Deployed Contracts** section in the **Deploy & Run Transactions** panel.
    - The contract instance (SingleNFT) is listed with its unique address (e.g., 0xD91...39138).
2. **Auto-Generated Interface**:
    - Remix automatically generates a user interface for all public and external functions defined in the contract.
    - Each function is displayed as a clickable button or input field (e.g., mint, listToken, buyToken, etc.), allowing users to interact with the contract directly without writing additional scripts.
3. **Interacting with Functions**:
    - You can test the contract's functionality by:
        - Inputting the required parameters into the corresponding fields.
        - Clicking the respective buttons to execute the functions (e.g., mint to create a token, buyToken to purchase it).
    - Results of function calls (e.g., transaction details, emitted events) appear in the lower panel.

4. **Transaction Logs**:
    - The lower panel displays logs for each executed transaction, including:

        - The sender’s address.
        - The function called and its parameters.
        - The transaction’s success or failure status.

5. **Next Steps**:
    - Using this interface, you can sequentially test all the contract's features, such as creating tokens, listing them, buying them, and verifying their state.

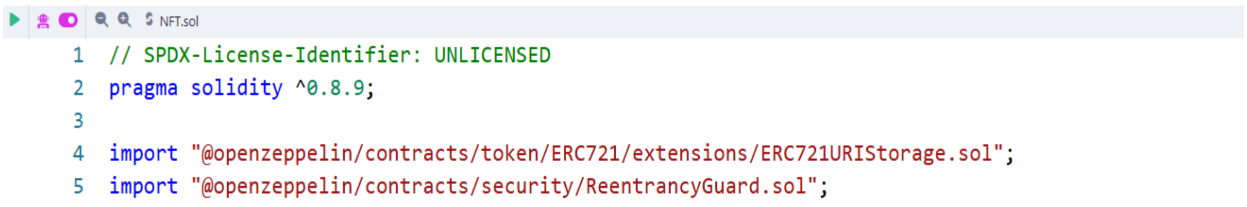
Transition to the Code Explanation

Now that we understand the setup and purpose of the SingleNFT contract, let’s dive into its structure. The contract is built using the **OpenZeppelin** libraries, which provide reliable and secure tools for creating smart contracts. This ensures the contract follows the widely accepted standards for NFTs and includes built-in protections.

In the following sections, we’ll break down the code step by step to explain:

1. How the contract is structured.
2. The key libraries and features it uses.
3. How its functions, like minting and selling NFTs, work.

This step-by-step approach will help us understand the purpose of each part of the code.



```
1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity ^0.8.9;
3
4  import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
5  import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
```

1 - SPDX License Identifier

- **Purpose**: Indicates that the contract is **not open-source licensed** by specifying UNLICENSED.

2 - Solidity Version

- **Purpose**: Specifies the required **Solidity compiler version**.
- **Details**: The contract requires version 0.8.9 or higher, but strictly below 0.9.0.

Imported Libraries:

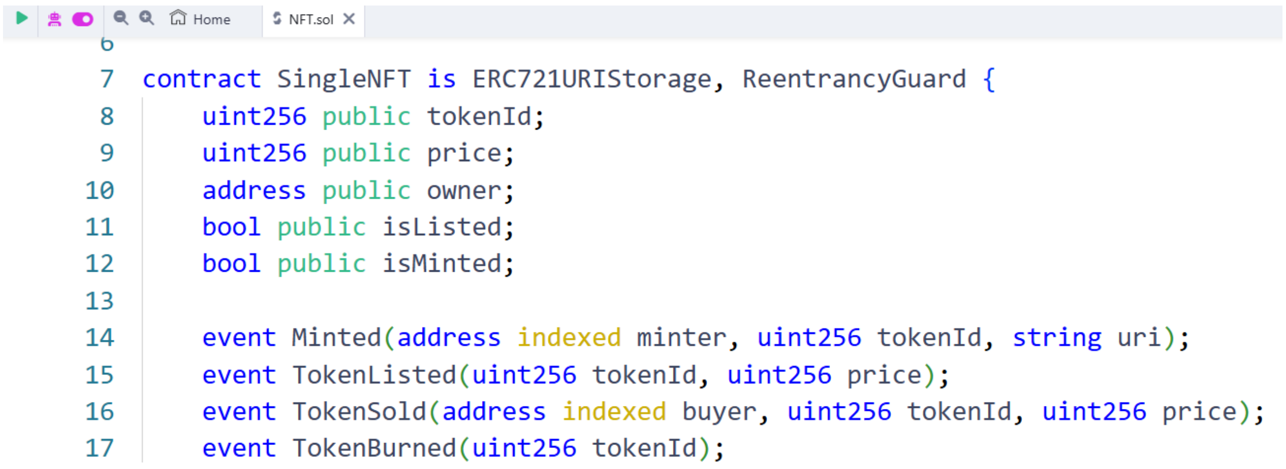
4 - ERC721URIStorage

- **Purpose**: Adds **URI storage** functionality for NFTs.
- **Details**: Allows each token to have unique metadata, which is commonly used for associating external resources (like images or JSON metadata) with tokens.

5 - ReentrancyGuard

- **Purpose**: Provides protection against **reentrancy attacks**.
- **Details**: Ensures functions cannot be recursively called before their initial execution is completed, which is crucial for secure handling of funds or sensitive operations.

The SingleNFT contract



```
 6
 7  contract SingleNFT is ERC721URIStorage, ReentrancyGuard {
 8      uint256 public tokenId;
 9      uint256 public price;
10      address public owner;
11      bool public isListed;
12      bool public isMinted;
13
14      event Minted(address indexed minter, uint256 tokenId, string uri);
15      event TokenListed(uint256 tokenId, uint256 price);
16      event TokenSold(address indexed buyer, uint256 tokenId, uint256 price);
17      event TokenBurned(uint256 tokenId);
```

7 - SingleNFT Contract

- Inherits metadata functionality from **ERC721URIStorage**, enabling the storage and management of metadata for NFTs.
- Incorporates protection against reentrancy attacks through **ReentrancyGuard**, ensuring safe handling of sensitive functions like token purchases.

Variables:

8 - tokenId

- **Purpose**: Serves as a unique identifier for the NFT.
- **Details**: Starts with a value of 1 and represents the token managed by the contract.

9 - price

- **Purpose**: Specifies the price of the token in **Wei**.
- **Details**: This value is set when the token is listed for sale.

10 - owner

- **Purpose**: Tracks the address of the current token owner.
- **Details**: Initially assigned to the deployer's address when the contract is deployed.

11 - isListed

- **Purpose**: Indicates whether the token is currently available for sale.
- **Details**: A boolean variable (true if listed for sale, false otherwise).

12 - isMinted

- **Purpose**: Tracks whether the token has been minted.
- **Details**: A boolean variable (true once minted, false otherwise).

Events:

14 - Minted

- **Purpose**: Logs the creation of a token.
- **Details**: Includes:
    - The address of the minter.
    - The token ID.
    - The token's metadata URI.

15 - TokenListed

- **Purpose**: Logs when a token is listed for sale.
- **Details**: Includes:
    - The token ID.
    - The price of the token.

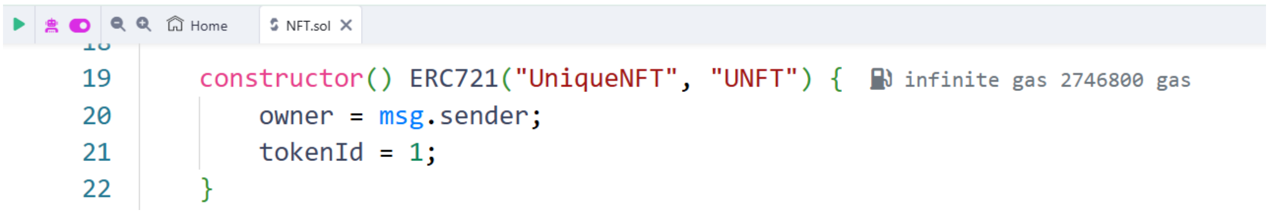
16 - TokenSold

- **Purpose**: Logs the successful purchase of a token.
- **Details**: Includes:
    - The buyer's address.
    - The token ID.
    - The price of the token.

17 - TokenBurned

- **Purpose**: Logs when a token is permanently removed from the blockchain.
- **Details**: Includes:
    - The token ID.

Constructor



```
19    constructor() ERC721("UniqueNFT", "UNFT") {  infinite gas 2746800 gas
20        owner = msg.sender;
21        tokenId = 1;
22    }
```

Function Details:

19 - constructor()

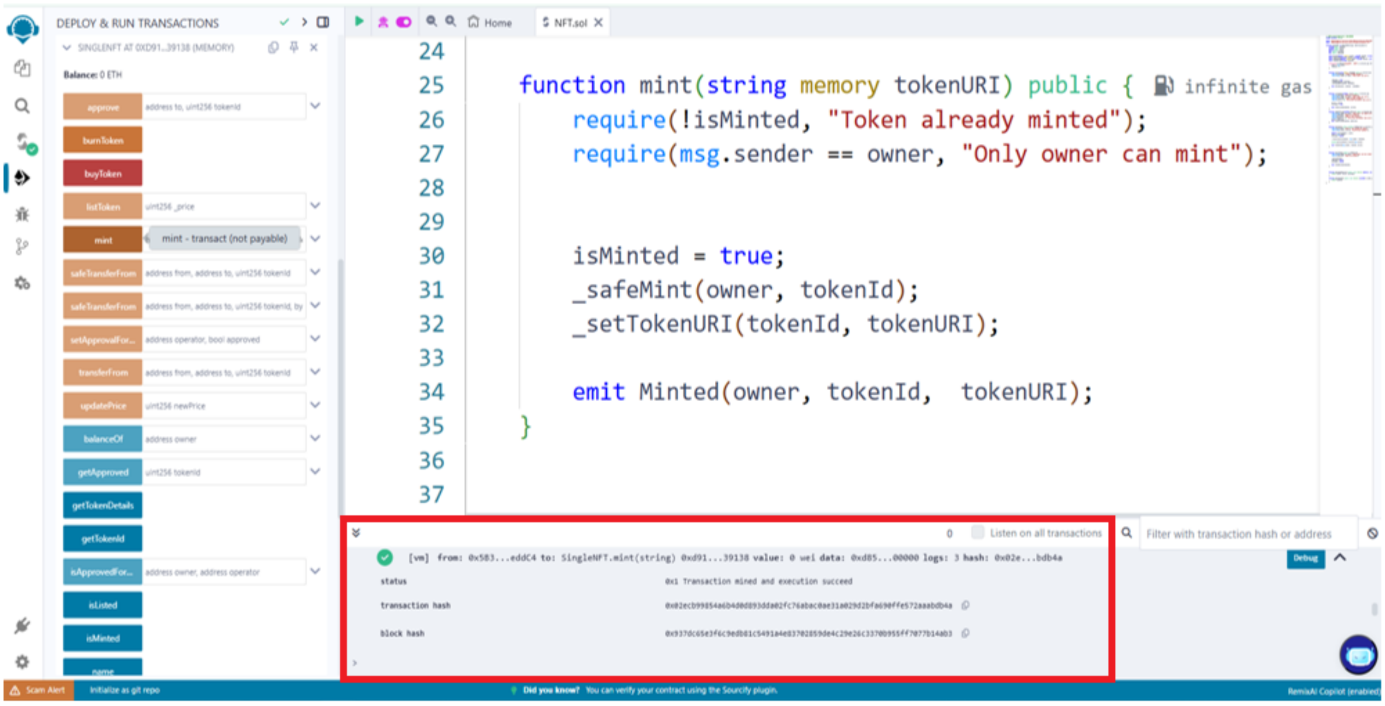
- **Purpose**: Initializes the contract with the collection name "UniqueNFT" and the symbol "UNFT".

- **Functionality**:

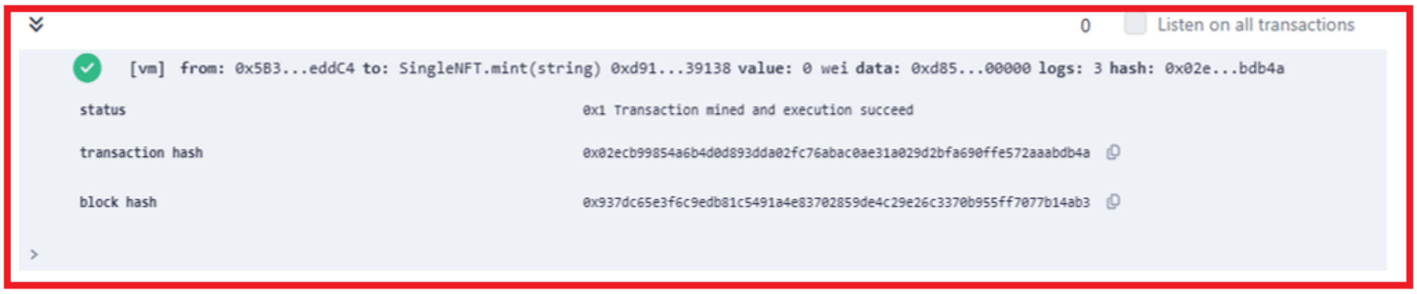
    - Sets the **Owner**: The deployer's address (msg.sender) is assigned as the owner of the contract.

    - Assigns tokenId: A fixed value of 1 is assigned to tokenId, making it the initial identifier for the token managed by the contract.

Transaction Mint Result:



```
24
25      function mint(string memory tokenURI) public {
26          require(!isMinted, "Token already minted");
27          require(msg.sender == owner, "Only owner can mint");
28
29
30          isMinted = true;
31          _safeMint(owner, tokenId);
32          _setTokenURI(tokenId, tokenURI);
33
34          emit Minted(owner, tokenId,  tokenURI);
35      }
36
37
```



```
0    Listen on all transactions
[vm] from: 0x5B3...eddC4 to: SingleNFT.mint(string) 0xd91...39138 value: 0 wei data: 0xd85...00000 logs: 3 hash: 0x02e...bdb4a
status               0x1 Transaction mined and execution succeed
transaction hash     0x02ecb99854a6b4d0d893dda02fc76abac0ae31a029d2bfa690ffe572aaabdb4a
block hash           0x937dc65e3f6c9edb81c5491a4e83702859de4c29e26c3370b955ff7077b14ab3
```

Step 1: Execute the mint Function

- Click the **mint** button to create a new token.
- Provide the required tokenURI as input, which specifies the metadata for the token.

Step 2: View Transaction Result

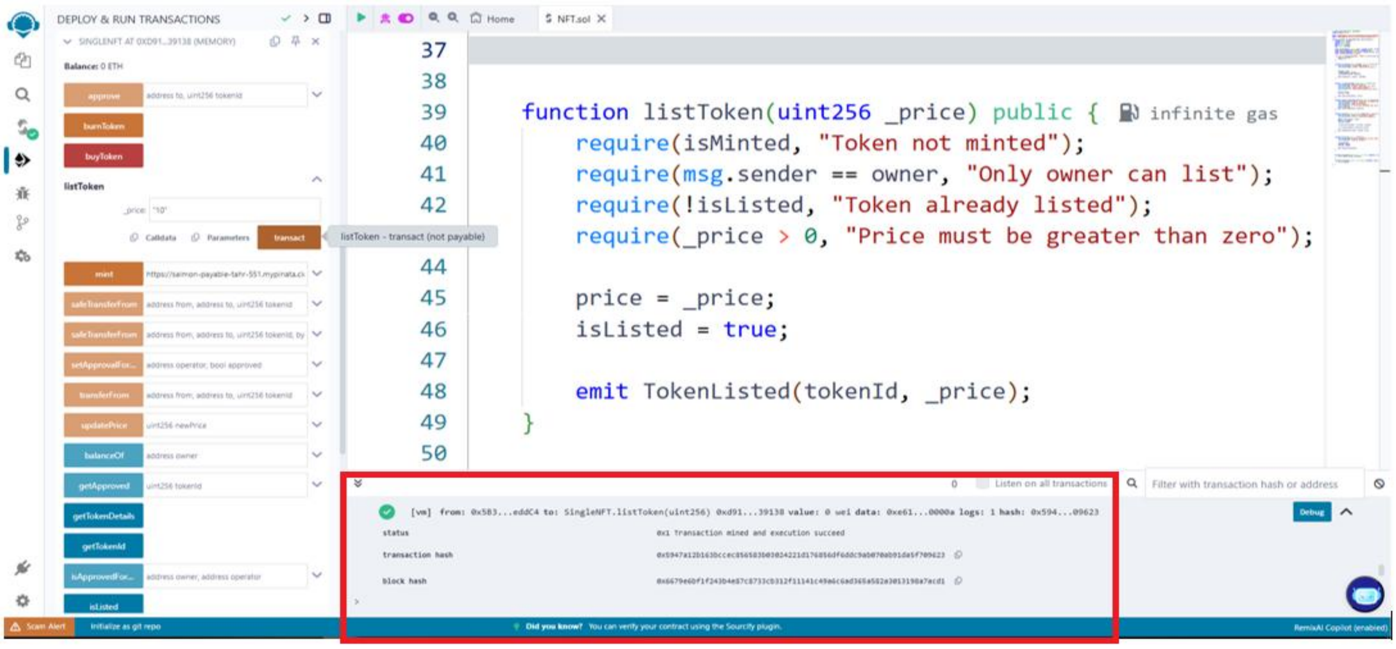
- After clicking the button, the transaction is processed, and a status message appears in the bottom panel, showing:
    - **Status**: Indicates whether the transaction succeeded (e.g., "transaction mined and execution succeed").
    - **Transaction Hash**: A unique identifier for the transaction.
    - **Block Hash**: The hash of the block that included this transaction.
    - **Logs**: The emitted Minted event, confirming the token's creation. The event includes:
        - The **Owner's Address**.
        - The **Token ID**.
        - The **Token URI**.

Function Details:

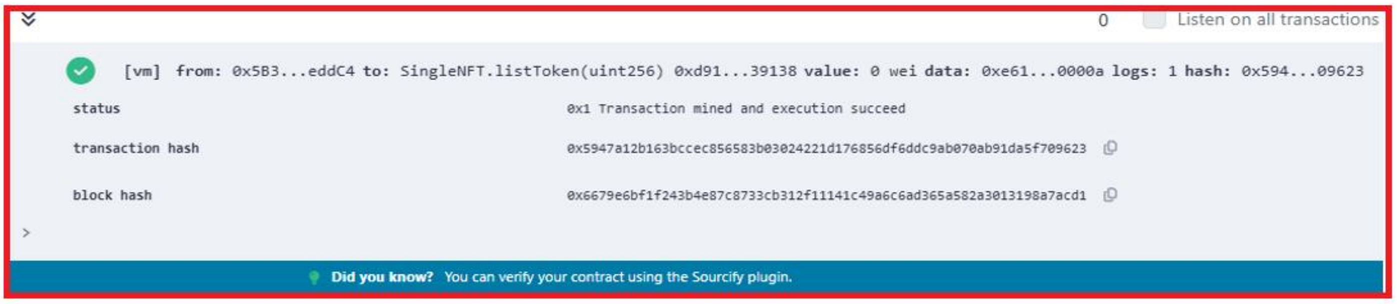
25 - mint(string memory tokenURI)

- **Purpose**: Mints a new token with a unique ID (tokenId) and sets its metadata URI (tokenURI).
- **Conditions**:
  - The token must not have been minted before (isMinted must be false).
  - Only the contract owner (owner) can execute the function.
- **Process**:
  - **30** - Marks the token as minted by setting isMinted to true.
  - **31** - Calls _safeMint(owner, tokenId) to safely mint the token and assign it to the owner's address.
  - **32** - Calls _setTokenURI(tokenId, tokenURI) to associate the token with the provided metadata URI.
  - **34** - Emits the Minted event, logging:
    - The **Owner's Address**.
    - The **Token ID**.
    - The **Token URI**.

listToken Function in the Interface:



```
function listToken(uint256 _price) public {
    require(isMinted, "Token not minted");
    require(msg.sender == owner, "Only owner can list");
    require(!isListed, "Token already listed");
    require(_price > 0, "Price must be greater than zero");

    price = _price;
    isListed = true;

    emit TokenListed(tokenId, _price);
}
```



```
0    Listen on all transactions

[vm] from: 0x5B3...eddC4 to: SingleNFT.listToken(uint256) 0xd91...39138 value: 0 wei data: 0xe61...0000a logs: 1 hash: 0x594...09623
status                              0x1 Transaction mined and execution succeed
transaction hash                    0x5947a12b163bccec856583b03024221d176856df6ddc9ab070ab91da5f709623
block hash                          0x6679e6bf1f243b4e87c8733cb312f11141c49a6c6ad365a582a3013198a7acd1
>
```

Did you know? You can verify your contract using the Sourcify plugin.

Step 1: Enter the Token Price

- In the **_price** input field, enter the desired price for the token.
  Example: Enter "10".

Step 2: Execute the listToken Function

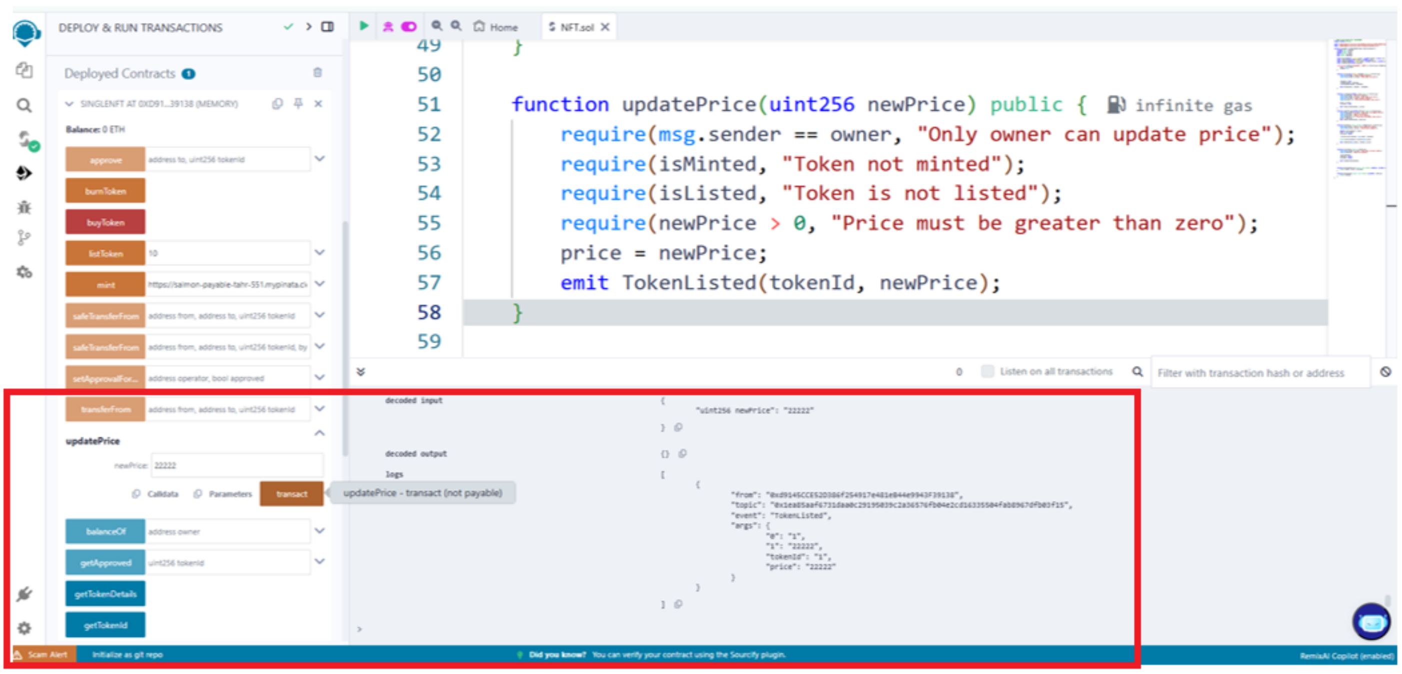
- Click the **listToken** button to list the token for sale.
- The result of the execution will appear in the bottom panel, including:
    - **Status**: Indicates whether the transaction succeeded.
    - **Transaction Hash**: A unique identifier for the transaction.
    - **Logs**: Emitted events confirming the successful listing of the token.

Function Details:

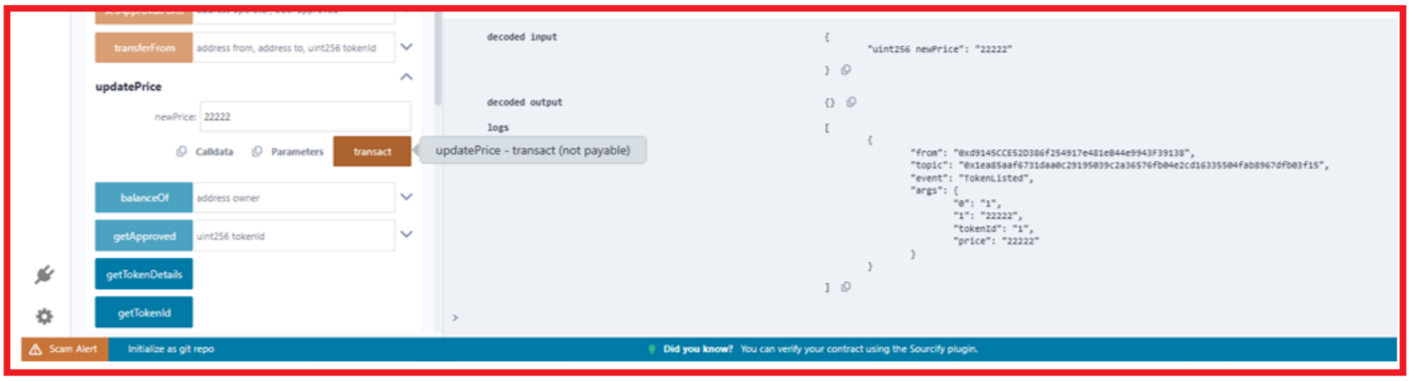
39 - listToken(uint256 _price)

- **Purpose**: Lists the token for sale by setting its price and marking it as listed.
- **Conditions**:
    - **40** - The token has already been minted (isMinted must be true).
    - **41** - Only the owner can list the token for sale (msg.sender == owner).
    - **42** - The token is not already listed (isListed must be false).
    - **43** - The price must be greater than zero (_price > 0).
- **Process**:
    - **45** - Assigns the specified _price to the price variable.
    - **46** - Sets isListed to true, indicating that the token is now available for sale.
    - **48** - Emits the TokenListed event, logging:
        - The **Token ID**.
        - The **Price**.

updatePrice Function in the Interface:



```
49      }
50
51      function updatePrice(uint256 newPrice) public {   infinite gas
52          require(msg.sender == owner, "Only owner can update price");
53          require(isMinted, "Token not minted");
54          require(isListed, "Token is not listed");
55          require(newPrice > 0, "Price must be greater than zero");
56          price = newPrice;
57          emit TokenListed(tokenId, newPrice);
58      }
59
```



```
decoded input     {
                      "uint256 newPrice": "22222"
                  }

decoded output    {}

logs              [
                      {
                          "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",
                          "topic": "0x1ea85aaf6731daa0c29195039c2a36576fb04e2cd16335504fab8967dfb03f15",
                          "event": "TokenListed",
                          "args": {
                              "0": "1",
                              "1": "22222",
                              "tokenId": "1",
                              "price": "22222"
                          }
                      }
                  ]
```

Step 1: Enter the New Price

- In the **newPrice** input field, enter the desired new price for the token.
  Example: Enter "22222".

Step 2: Execute the updatePrice Function

- Click the **updatePrice** button to update the token's price.
- The result of the execution will appear in the bottom panel, including:
  - **Decoded Input**: Displays the input provided (newPrice).
  - **Logs**: Shows the emitted TokenListed event, confirming the price update. The event includes:
    - **Token ID**: The ID of the token.
    - **New Price**: The updated price of the token.

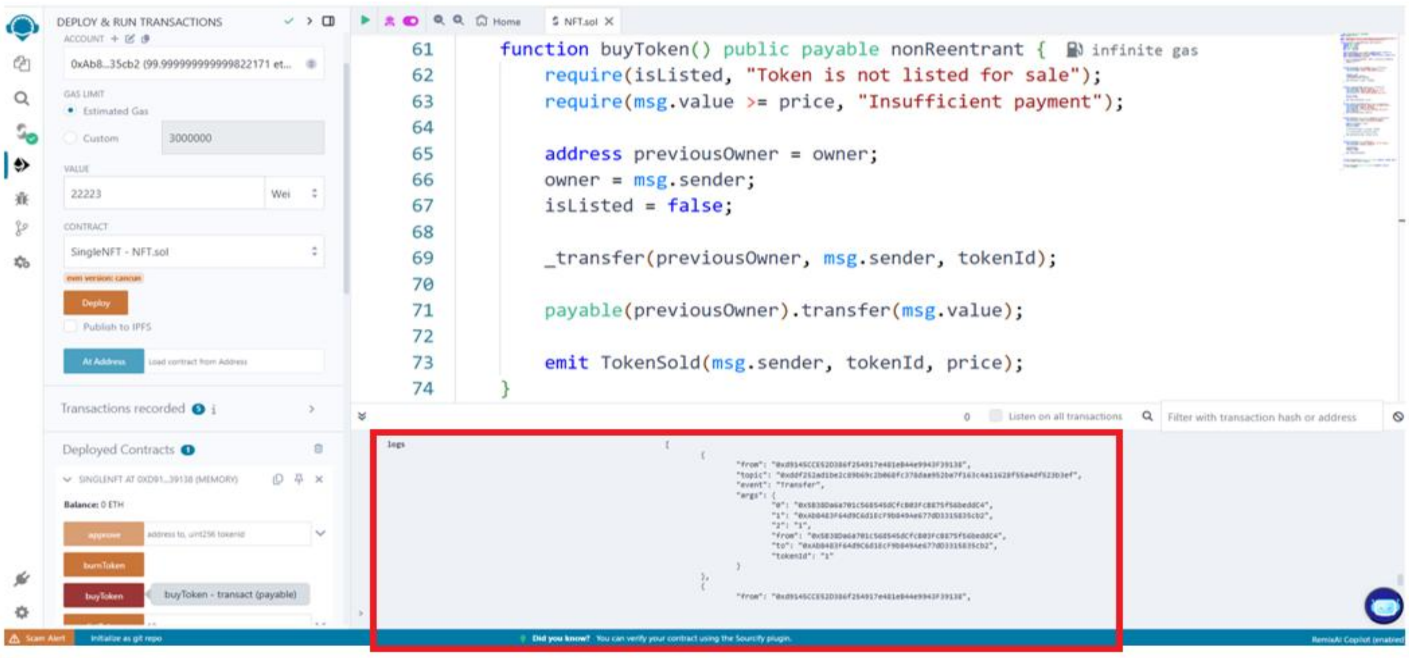
Function Details:

51 - updatePrice(uint256 newPrice)

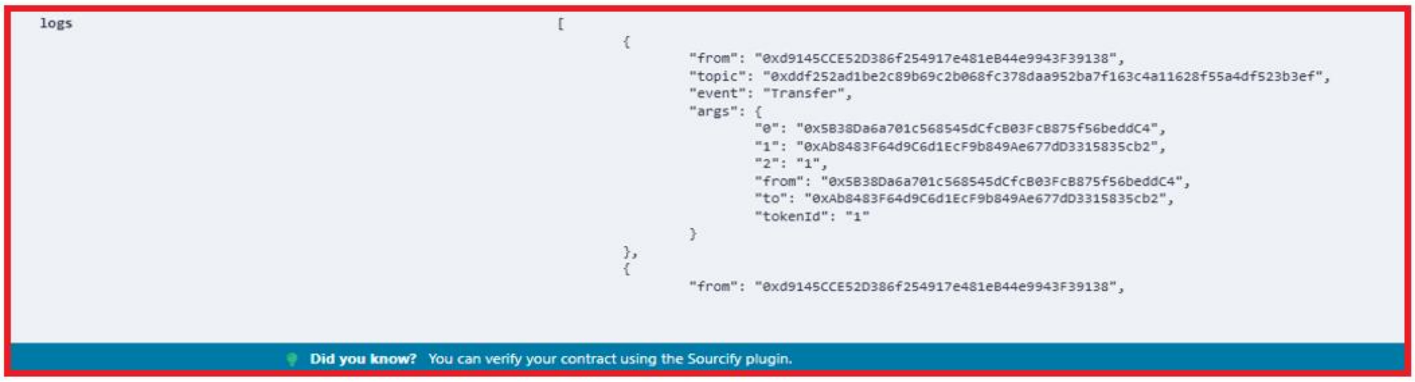
- **Purpose**: Updates the price of the listed token.
- **Conditions**:
  - **52** - Only the owner can update the price (msg.sender == owner).

    - **53** - The token has already been minted (isMinted must be true).
    - **54** - The token is currently listed for sale (isListed must be true).
    - **55** - The new price must be greater than zero (newPrice > 0).
- **Process**:
    - **56** - Assigns the specified newPrice to the price variable.
    - **57** - Emits the TokenListed event, logging:
        - The **Token ID**.
        - The **Updated Price**.

buyToken function in the Interface:


DEPLOY & RUN TRANSACTIONS
ACCOUNT
0xAb8...35cb2 (99.999999999999822171 et...
GAS LIMIT
Estimated Gas
Custom 3000000
VALUE
22223 Wei
CONTRACT
SingleNFT - NFT.sol
Deploy
Publish to IPFS
At Address
Load contract from Address
Transactions recorded 3
Deployed Contracts 1
SINGLENFT AT 0XD91...39138 (MEMORY)
Balance: 0 ETH
approve
burnToken
buyToken
buyToken - transact (payable)
Home
NFT.sol
Listen on all transactions
Filter with transaction hash or address
Did you know? You can verify your contract using the Sourcify plugin.

```
function buyToken() public payable nonReentrant {   infinite gas
    require(isListed, "Token is not listed for sale");
    require(msg.value >= price, "Insufficient payment");

    address previousOwner = owner;
    owner = msg.sender;
    isListed = false;

    _transfer(previousOwner, msg.sender, tokenId);

    payable(previousOwner).transfer(msg.value);

    emit TokenSold(msg.sender, tokenId, price);
}
```



```
logs                                   [
                                          {
                                                "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",
                                                "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
                                                "event": "Transfer",
                                                "args": {
                                                        "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
                                                        "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                                                        "2": "1",
                                                        "from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
                                                        "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                                                        "tokenId": "1"
                                                }
                                          },
                                          {
                                                "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",
```

Did you know? You can verify your contract using the Sourcify plugin.

Step 1: Switch the Active Account

- In the **Account** dropdown, switch to a different address from the one that owns the NFT (e.g., select an account with a sufficient balance).

Step 2: Enter the Payment Amount

- In the **Value** input field, enter the required payment amount (equal to or greater than the token's price). Example: Enter "22223" Wei.

Step 3: Execute the buyToken Function

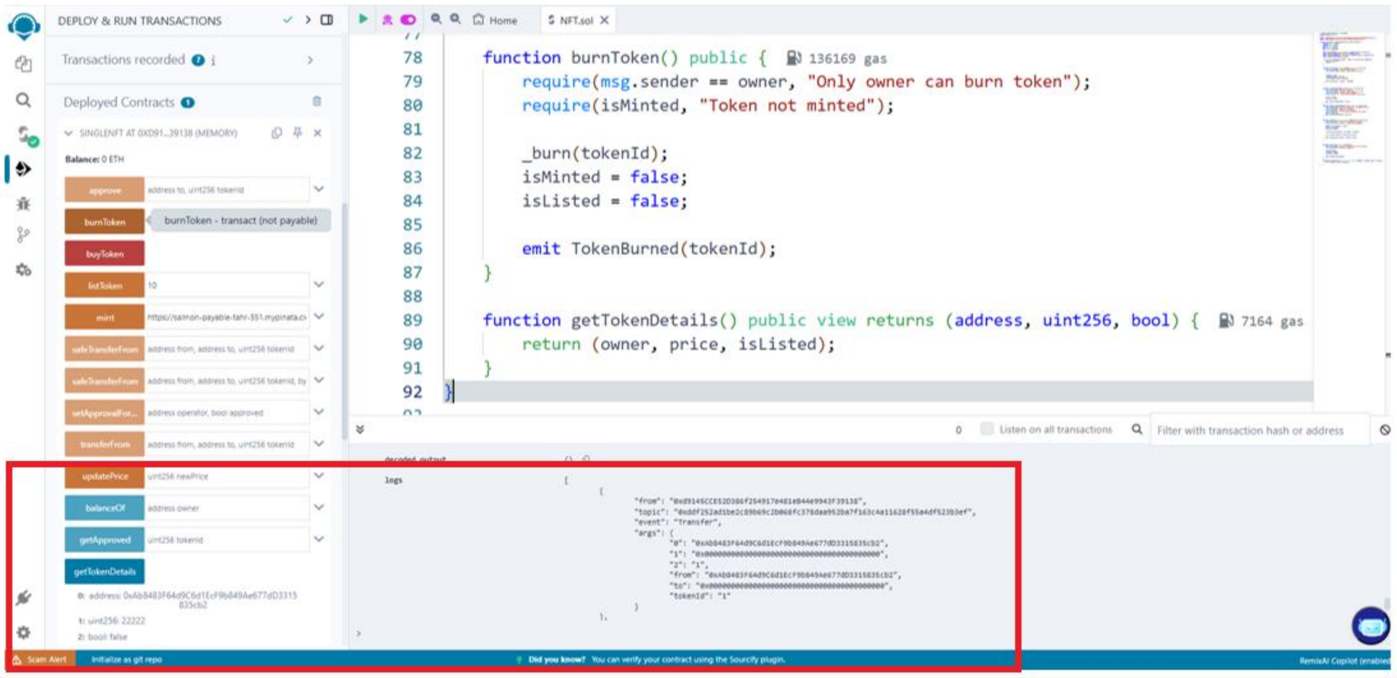
- Click the **buyToken** button to purchase the NFT.
- The result of the execution will appear in the bottom panel, showing:
    - **Event Logs**: The TokenSold event, which includes the buyer's address, token ID, and price.
    - **Transfer Details**: Confirmation of the token's transfer from the previous owner to the buyer.

Function Details:

61 - buyToken()

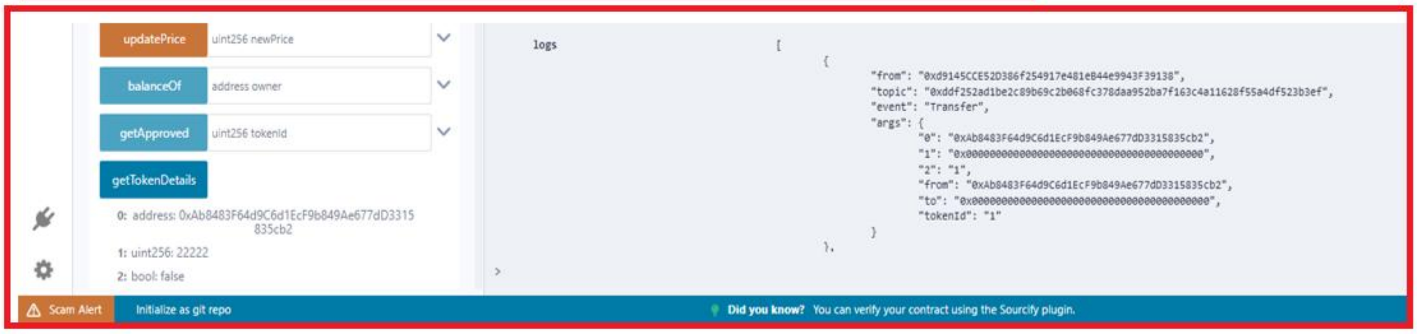
- **Purpose**: Allows a user to purchase a listed NFT by sending the required payment.
- **Conditions**:
    - **62** - The token must be listed for sale (isListed must be true).
    - **63** - The buyer must send a payment amount (msg.value) equal to or greater than the token's price (price).
- **Process**:
    - **65** - previousOwner: Stores the address of the current owner before transferring the token.
    - **66** - Updates the owner to the buyer's address (msg.sender) and marks the token as no longer listed (isListed = false).
    - **69** - Transfers the token from the previous owner to the buyer using _transfer.
    - **71** - Transfers the payment (msg.value) to the previous owner.
    - **73** - Emits the TokenSold event, logging:
        - The buyer's address.
        - The token ID.
        - The price.

Execution Flow in the Interface:



DEPLOY & RUN TRANSACTIONS
Transactions recorded
Deployed Contracts
SINGLENFT AT 0XD91...39138 (MEMORY)
Balance: 0 ETH
approve
burnToken
burnToken - transact (not payable)
buyToken
listToken
mint
safeTransferFrom
setApprovalFor...
transferFrom
updatePrice
balanceOf
getApproved
getTokenDetails
Home
NFT.sol
Listen on all transactions
Filter with transaction hash or address
Scam Alert
Initialize as git repo
Did you know? You can verify your contract using the Sourcify plugin.
RemixAI Copilot (enabled)

```
78     function burnToken() public {   136169 gas
79         require(msg.sender == owner, "Only owner can burn token");
80         require(isMinted, "Token not minted");
81
82         _burn(tokenId);
83         isMinted = false;
84         isListed = false;
85
86         emit TokenBurned(tokenId);
87     }
88
89     function getTokenDetails() public view returns (address, uint256, bool) {   7164 gas
90         return (owner, price, isListed);
91     }
92 }
```



```
0: address: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
1: uint256: 22222
2: bool: false
```

```
logs        [
                {
                    "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",
                    "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
                    "event": "Transfer",
                    "args": {
                        "0": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                        "1": "0x0000000000000000000000000000000000000000",
                        "2": "1",
                        "from": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
                        "to": "0x0000000000000000000000000000000000000000",
                        "tokenId": "1"
                    }
                },
```

Step 1: Call the getTokenDetails Function

- Click the **getTokenDetails** button.
- The decoded output will display:
  - **Owner Address**: The current owner of the token.
  - **Price**: The price of the token if it is listed for sale.
  - **Listing Status**: Whether the token is currently listed (true or false).

Step 2: Execute the burnToken Function

- Click the **burnToken** button to permanently destroy the token.
- The transaction will emit a TokenBurned event, confirming the destruction of the token.
- After burning:
  - The token is no longer available.
  - Its status (isMinted and isListed) is reset to false.

Function Details:

78 - burnToken()

- **Purpose**: Permanently destroys the token.
- **Conditions**:
  - **79** - Only the owner can burn the token (msg.sender == owner).
  - **80** - The token has already been minted (isMinted must be true).
- **Process**:

    - **82** - Calls _burn(tokenId) to remove the token from the blockchain.
    - **83** - Resets isMinted and isListed to false, reflecting that the token no longer exists.
    - **86** - Emits the TokenBurned event, logging the token ID that was destroyed.

89 - getTokenDetails()

- **Purpose**: Returns the current details of the token, including:
    - **Owner Address**: The address of the current owner.
    - **Price**: The token's price if it is listed for sale.

  Listing Status: Whether the token is listed for sale (isListed).

Conclusion

The SingleNFT contract not only served as an example of implementing NFTs but also provided us with a deeper understanding of how smart contracts work. By leveraging **OpenZeppelin**, we explored the key aspects of building secure and standardized contracts. Here are the main takeaways:

1. **Understanding Smart Contracts**:
    - We learned how smart contracts interact with the blockchain, process data, and manage state.
    - We built functions for critical operations such as minting, listing, selling, and burning tokens.
    - We explored essential security practices, including protection against reentrancy attacks using ReentrancyGuard.
2. **Practical Application**:
    - We gained hands-on experience interacting with the contract through Remix IDE.
    - We observed how to create and process transactions, reviewing their results through the interface.
    - This exercise highlighted how contracts can be tested and adapted in real-time.
3. **Modularity and Scalability**:
    - The contract demonstrated how ready-made standards, such as ERC721, can be utilized and extended with custom functionality.
    - We saw how smart contracts can be adapted to new requirements, such as adding royalties or advanced marketplace features.

This project not only helped us understand the fundamental functions of NFTs but also gave us valuable insight into the core concepts of smart contracts. SingleNFT stands as an excellent example of combining theory and practice to build real-world blockchain solutions.